

Capítulo VINTE E QUATRO

NIO.2

Objetivos do Exame

- Usar a interface Path para operar em caminhos de arquivos e diretórios.
 - Usar a classe Files para verificar, ler, excluir, copiar, mover e gerenciar metadados de um arquivo ou diretório.
-

NIO.2

No último capítulo, revisamos as classes do pacote `java.io`.

Nas primeiras versões do Java, este pacote, especialmente a classe `File`, fornecia suporte para operações de arquivos. No entanto, ele tinha alguns problemas, como falta de funcionalidade e suporte limitado a atributos de arquivos.

Por essa razão, o Java 1.4 introduziu a API NIO (Non-blocking Input/Output) no pacote `java.nio`, implementando novas funcionalidades como canais, buffers e novos conjuntos de caracteres.

No entanto, essa API não resolveu completamente os problemas com o pacote `java.io`, então, no Java 7, a API NIO.2 foi adicionada no pacote `java.nio.file` (na verdade, como este é um novo pacote, o NIO.2 não é uma atualização da API NIO; além disso, elas se concentram em coisas diferentes).

O NIO.2 fornece melhor suporte para acesso a arquivos e ao sistema de arquivos, links simbólicos, interoperabilidade, exceções, entre outros.

As principais classes de `java.nio.file`, `Path`, `Paths` e `Files`, têm como objetivo fornecer uma maneira mais fácil de trabalhar com arquivos e substituir a classe `java.io.File`.

Essas classes serão o foco deste capítulo. Vamos começar com `Path` e `Paths`.

A interface Path

No capítulo anterior, também revisamos conceitos como sistemas de arquivos e caminhos.

Bem, a interface `Path` define um objeto que representa o caminho até um arquivo ou diretório.

Quando você pensa no fato de que um caminho varia entre diferentes sistemas de arquivos, faz sentido que `Path` seja uma interface. Graças a isso, o Java lida de forma transparente com diferentes implementações entre plataformas.

Por exemplo, aqui estão algumas diferenças entre sistemas baseados em Windows e sistemas baseados em Unix:

- Sistemas baseados em Windows não diferenciam maiúsculas de minúsculas, enquanto sistemas baseados em Unix diferenciam.
- Em sistemas baseados em Windows, os caminhos são separados por barras invertidas. Em sistemas baseados em Unix, por barras normais.
- Em sistemas baseados em Windows, o caminho raiz é uma letra de unidade (geralmente `c:`). Em sistemas baseados em Unix, é uma barra normal (`/`).
- Por causa disso, em sistemas baseados em Windows, um caminho absoluto começa com uma letra de unidade (como `c:\temp\file.txt`). Em sistemas baseados em Unix, começa com uma barra (como `/temp/file.txt`).

Como `Path` é uma interface e o Java lida com suas implementações, temos que usar uma classe utilitária para criar instâncias de `Path`.

`java.nio.file.Paths` é essa classe. Ela fornece dois métodos para criar um objeto `Path`:

```
java Copiar Editar

static Path get(String first, String... more)
static Path get(URI uri)
```

Tenha muito cuidado com os nomes:

- Path é a interface com métodos para trabalhar com caminhos.
- Paths é a classe com métodos estáticos para criar um objeto Path.

Com a primeira versão de Paths.get(), você pode criar um objeto Path dessas maneiras:

```
java Copiar Editar

// Com um caminho absoluto no Windows
Path pathWin = Paths.get("c:\\temp\\file.txt");

// Com um caminho absoluto no Unix
Path pathUnix = Paths.get("/temp/file.txt");

// Com um caminho relativo
Path pathRelative = Paths.get("file.txt");

// Usando o parâmetro varargs (o separador é inserido automaticamente)
Path pathByParts = Paths.get("c:", "temp", "file.txt");
```

Com a segunda versão, você tem que usar uma instância de java.net.URI. Como estamos trabalhando com arquivos, o esquema da URI deve ser file://:

```
java Copiar Editar

try {
    Path fileURI = Paths.get(new URI("file:///c:/temp/file.txt"));
} catch (URISyntaxException e) {
    // Esta exceção verificada é lançada pelo construtor URI
}
```

Se você não quiser capturar URISyntaxException, pode usar o método estático URI.create(String). Ele encapsula a exceção URISyntaxException em uma IllegalArgumentException (uma subclasse de RuntimeException):

```
java Copiar Editar

Path fileURI = Paths.get(URI.create("file:///c:/temp/file.txt"));
```

Observe as três barras. file:/// representa um caminho absoluto (o esquema file:// mais uma barra para o diretório raiz). Podemos testar isso com a ajuda do método toAbsolutePath(), que retorna a representação absoluta de um objeto Path:

```
java Copiar Editar

Path fileURI = Paths.get(URI.create("file:///file.txt"));
System.out.println(fileURI.toAbsolutePath());
```

Isso imprimirá:

```
cpp Copiar Editar

C:\file.txt // em sistemas baseados em Windows
/file.txt   // Ou em sistemas baseados em Unix
```

Também podemos criar um Path a partir de um File e vice-versa:

```
java Copiar Editar

File file = new File("/file.txt");
Path path = file.toPath();

path = Paths.get("/file.txt");
file = path.toFile();
```

E só para deixar claro que a instância de Path é dependente do sistema, deixe-me dizer que Paths.get() é na verdade equivalente a:

```
java Copiar Editar

Path path = FileSystems.getDefault().getPath("c://temp");
```

Como você pode ver nos exemplos, a representação do caminho absoluto de um objeto Path possui um componente raiz (ou c:\ ou /) e uma sequência de nomes separados por uma barra (normal ou invertida).

Esses nomes representam os diretórios necessários para navegar até o arquivo ou diretório de destino. O último nome na sequência representa o nome do arquivo ou diretório de destino.

Por exemplo, os elementos do caminho c:\temp\dir1\file.txt (ou seu equivalente no Unix, /temp/dir1/file.txt) são:

- Raiz: c:\ (ou /)
- Nome 1: temp
- Nome 2: dir1
- Nome 3: file.txt

O objeto Path possui alguns métodos para obter essas informações. Exceto por toString() e getNameCount(), cada um desses métodos retorna um objeto Path:

```
java Copiar Editar

Path path = Paths.get("C:\\temp\\dir1\\file.txt");
// Ou Path path = Paths.get("/temp/dir1/file.txt");

System.out.println("toString(): " + path.toString());
System.out.println("getFileName(): " + path.getFileName());
System.out.println("getNameCount(): " + path.getNameCount());
// Os índices começam em zero
System.out.println("getName(0): " + path.getName(0));
System.out.println("getName(1): " + path.getName(1));
System.out.println("getName(2): " + path.getName(2));
// subpath(beginIndex, endIndex) vai de beginIndex até endIndex - 1
System.out.println("subpath(0,2): " + path.subpath(0,2));
System.out.println("getParent(): " + path.getParent());
System.out.println("getRoot(): " + path.getRoot());
```

A saída:

```
scss Copiar Editar

toString(): C:\temp\dir1\file.txt // Ou /temp/dir1/file.txt
getFileName(): file.txt
getNameCount(): 3
getName(0): temp
getName(1): dir1
getName(2): file.txt
subpath(0,2): temp\dir1 // Ou temp/dir1
getParent(): C:\temp\dir1 // Ou /temp/dir1
getRoot(): C:\ // Ou /
```

Passar um índice inválido para `getName()` e `subpath()` lançará uma `IllegalArgumentException` (uma `RuntimeException`).

Se o caminho for especificado como relativo (e assumindo que esse código seja executado a partir do diretório `c:\temp`):

```
java Copiar Editar

Path path = Paths.get("dir1\\file.txt"); // Ou dir1/file.txt
System.out.println("toString(): " + path.toString());
System.out.println("getFileName(): " + path.getFileName());
System.out.println("getNameCount(): " + path.getNameCount());
System.out.println("getName(0): " + path.getName(0));
System.out.println("getName(1): " + path.getName(1));
System.out.println("subpath(0,2): " + path.subpath(0,2));
System.out.println("getParent(): " + path.getParent());
System.out.println("getRoot(): " + path.getRoot());
```

A saída:

```
scss Copiar Editar

toString(): dir1\file.txt // Ou dir1/file.txt
getFileName(): file.txt
getNameCount(): 2
getName(0): dir1
getName(1): file.txt
subpath(0,2): dir1\file.txt // Ou dir1/file.txt
getParent(): dir1
getRoot(): null
```

Ao trabalhar com caminhos, você pode usar:

- `.` para se referir ao diretório atual
- `..` para se referir ao diretório pai

Por exemplo:

```
java Copiar Editar

// refere-se a /temp/file.txt
Path p1 = Paths.get("/temp../file.txt");
// refere-se a /temp//file.txt
Path p2 = Paths.get("/temp/dir1../file.txt");
```

Nesses casos, você pode usar o método `normalize()` para remover redundâncias como `.` e `..` (em outras palavras, para “normalizar” o caminho):

```
java Copiar Editar

Path path = Paths.get("/temp/dir1/../file.txt");
System.out.println(path); // /temp/dir1/../file.txt
Path path2 = path.normalize();
System.out.println(path2); // /temp/file.txt
```

Este método não acessa o sistema de arquivos para saber se um arquivo existe, então remover `..` e um nome anterior de um caminho pode resultar em um caminho que não referencia mais o arquivo original. Isso pode acontecer quando esse nome anterior for um link simbólico (uma referência a outro arquivo).

É melhor usar o método `toRealPath()`:

```
java Copiar Editar

Path toRealPath(LinkOption... options) throws IOException
```

Este método faz o seguinte:

- Se `LinkOption.NOFOLLOW_LINKS` for passado como argumento, links simbólicos não são seguidos (por padrão, são).
- Se o caminho for relativo, ele retorna um caminho absoluto.
- Retorna um `Path` com elementos redundantes removidos (se houver).

Podemos combinar dois caminhos. Há dois casos.

Primeiro caso: Se tivermos um caminho absoluto e quisermos combiná-lo com um segundo caminho que não tenha um elemento raiz (um caminho parcial), o segundo caminho é acrescentado:

```
java Copiar Editar

Path path = Paths.get("/temp");
System.out.println(path.resolve("newDir")); // /temp/newDir
```

Segundo caso: Se tivermos um caminho parcial ou relativo e quisermos combiná-lo com um caminho absoluto, esse caminho absoluto é retornado:

```
java Copiar Editar

Path path = Paths.get("newDir");
System.out.println(path.resolve("/temp")); // /temp
```

O método `relativize()` também é interessante.

`path1.relativize(path2)` é como dizer: “me diga um caminho que mostre como ir de `path1` até `path2`”.

Por exemplo, se estamos no diretório `/temp` e queremos ir para `/temp/dir1/subdir`, precisamos ir primeiro para `dir1` e depois para `subdir`:

```
java Copiar Editar

Path path1 = Paths.get("temp");
Path path2 = Paths.get("temp/dir1/file.txt");
Path path1ToPath2 = path1.relativize(path2); // dir1/file.txt
```

Se os caminhos representarem dois caminhos relativos sem nenhuma outra informação, eles são considerados irmãos, então é preciso ir ao diretório pai e depois ao outro diretório:

```
java Copiar Editar

Path path1 = Paths.get("dir1");
Path path1ToPath2 = path1.relativeTo(Paths.get("dir2")); // ../dir2
```

Observe que ambos os exemplos usam caminhos relativos.

Se um dos caminhos for absoluto, um caminho relativo não pode ser construído por falta de informações, e uma `IllegalArgumentException` será lançada.

Se ambos os caminhos forem absolutos, o resultado dependerá do sistema.

A interface `Path` implementa a interface `Iterable`, então você pode fazer algo assim:

```
java Copiar Editar

Path path = Paths.get("c:\\temp\\dir1\\file.txt");
for (Path name : path) {
    System.out.println(name);
}
```

A saída:

```
nginx Copiar Editar

temp
dir1
file.txt
```

A interface `Path` implementa `Comparable` e o método `equals()` para testar dois caminhos quanto à igualdade.

`compareTo()` compara dois caminhos lexicograficamente. Ele retorna:

- Zero se o argumento for igual ao caminho.
- Um valor menor que zero se este caminho for lexicograficamente menor que o argumento.
- Um valor maior que zero se este caminho for lexicograficamente maior que o argumento.

A implementação de `equals()` é dependente do sistema (por exemplo, ela não diferencia maiúsculas de minúsculas em sistemas Windows). No entanto, ela retorna `false` se o argumento não for um `Path` ou se pertencer a um sistema de arquivos diferente.

Além disso, os métodos `startsWith()` e `endsWith()` testam se um caminho começa ou termina com uma `String` (nesse caso, os métodos retornam `true` somente se a string representar um elemento real) ou com um `Path`. Dado:

```
java Copiar Editar

Path absPath = Paths.get("c:\\temp\\dir1\\file.txt");
Path relPath = Paths.get("temp\\dir1\\file.txt");
```

`boolean startsWith(Path other)`

```
java Copiar Editar

absPath.startsWith(Paths.get("c:\\temp\\file.txt")); // false
absPath.startsWith(Paths.get("c:\\temp\\dir1\\img.jpg")); // false
absPath.startsWith(Paths.get("c:\\temp\\dir1\\")); // true
absPath.startsWith(relPath); // false
```

boolean startsWith(String other)

```
java Copiar Editar

relPath.startsWith("t"); // false
relPath.startsWith("temp"); // true
relPath.startsWith("temp\\d"); // false
relPath.startsWith("temp\\dir1"); // true
```

boolean endsWith(Path other)

```
java Copiar Editar

absPath.endsWith("file.txt"); // true
absPath.endsWith("d:\\temp\\dir1\\file.txt"); // false
relPath.endsWith(absPath); // false
```

boolean endsWith(String other)

```
java Copiar Editar

relPath.endsWith("txt"); // false
relPath.endsWith("file.txt"); // true
relPath.endsWith("\\dir1\\file.txt"); // false
relPath.endsWith("dir1\\file.txt"); // true
```

Esses métodos não levam em conta separadores finais, então se tivermos o Path temp/dir1, invocar, por exemplo, endsWith() com dir1/ retorna true.

A classe Files

A classe java.nio.file.Files possui métodos estáticos para operações comuns em arquivos e diretórios. Em contraste com a classe java.io.File, todos os métodos de Files funcionam com objetos Path (então, não confunda File com Files).

Por exemplo, podemos verificar se um caminho realmente existe (ou não existe) com os métodos:

```
java Copiar Editar

static boolean exists(Path path, LinkOption... options)
static boolean notExists(Path path, LinkOption... options)
```

Se LinkOption.NOFOLLOW_LINKS estiver presente, links simbólicos não são seguidos (por padrão, são).

Podemos verificar se um caminho é legível (não é, se o arquivo não existir ou se a JVM não tiver privilégios para acessá-lo):

```
java Copiar Editar

static boolean isReadable(Path path)
```

Podemos verificar se um caminho é gravável (não é, se o arquivo não existir ou se a JVM não tiver privilégios para acessá-lo):

```
java Copiar Editar

static boolean isWritable(Path path)
```

Podemos verificar se um arquivo existe e é executável:

```
java Copiar Editar

static boolean isExecutable(Path path)
```

Ou até verificar se dois caminhos referem-se ao mesmo arquivo (útil se um dos caminhos representar um link simbólico). Se ambos os objetos Path forem iguais, esse método retorna true sem verificar se o arquivo existe:

```
java Copiar Editar

static boolean isSameFile(Path path, Path path2) throws IOException
```

Para ler um arquivo, podemos carregar o arquivo inteiro na memória (útil apenas para arquivos pequenos) com os métodos:

```
java Copiar Editar

static byte[] readAllBytes(Path path)
    throws IOException

static List<String> readAllLines(Path path)
    throws IOException

static List<String> readAllLines(Path path, Charset cs)
    throws IOException
```

Por exemplo:

```
java Copiar Editar

try {
    // Por padrão, usa StandardCharsets.UTF_8
    List<String> lines = Files.readAllLines(
        Paths.get("file.txt"));
    lines.forEach(System.out::println);
} catch (IOException e) { /** */ }
```

Ou, para ler um arquivo de forma eficiente:

```
java Copiar Editar

static BufferedReader newBufferedReader(Path path)
    throws IOException

static BufferedReader newBufferedReader(Path path, Charset cs)
    throws IOException
```


Por exemplo:

```
java Copiar Editar

Path path = Paths.get("/temp/dir1/files.txt");
// Por padrão, usa StandardCharsets.UTF_8

try (BufferedReader reader = Files.newBufferedReader(path,
    StandardCharsets.ISO_8859_1)) {
    String line = null;
    while((line = reader.readLine()) != null)
        System.out.println(line);
} catch (IOException e) { /** ... */ }
```

A classe Files possui dois métodos para excluir arquivos/diretórios.

```
java Copiar Editar

static void delete(Path path) throws IOException
```

Ele remove o arquivo/diretório ou lança uma exceção se algo falhar:

```
java Copiar Editar

try {
    Files.delete(Paths.get("/temp/dir1/file.txt"));
    Files.delete(Paths.get("/temp/dir1"));
} catch (NoSuchFileException nsfe) {
    // Se o arquivo/diretório não existir
} catch (DirectoryNotEmptyException dneee) {
    // Para excluir um diretório, ele deve estar vazio,
    // caso contrário, essa exceção é lançada
} catch (IOException ioe) {
    // Permissões de arquivo ou outros problemas
}
```

O segundo método é:

```
java Copiar Editar

static boolean deleteIfExists(Path path) throws IOException
```

Este método retorna true se o arquivo foi excluído ou false se o arquivo não pôde ser removido porque não existia; ou seja, ao contrário do primeiro método, ele **não** lança `NoSuchFileException` (mas ainda lança `DirectoryNotEmptyException` e `IOException` para outros problemas):

```
java Copiar Editar

try {
    Files.delete(Paths.get("/temp/dir1/file.txt"));
} catch (DirectoryNotEmptyException dneee) {
    // Para excluir um diretório, ele deve estar vazio
} catch (IOException ioe) {
    // Permissões de arquivo ou outros problemas
}
```

Para copiar arquivos/diretórios, temos o método:

```
java
static Path copy(Path source, Path target,
                  CopyOption... options) throws IOException
```

Ele retorna o caminho para o arquivo de destino, e ao copiar um diretório, seu conteúdo **não** será copiado.

Por padrão, a cópia falha se o arquivo de destino já existir. Além disso, atributos de arquivos não são copiados e, ao copiar um link simbólico, o alvo do link será copiado.

Podemos personalizar esse comportamento com os seguintes enums CopyOption:

- StandardCopyOption.REPLACE_EXISTING
Executa a cópia quando o destino já existe. Se o destino for um link simbólico, o próprio link é copiado. Se o destino for um diretório não vazio, uma FileAlreadyExistsException é lançada.
- StandardCopyOption.COPY_ATTRIBUTES
Copia os atributos do arquivo associado ao arquivo de origem para o destino. Os atributos exatos suportados dependem do sistema de arquivos e da plataforma, exceto last-modified-time, que é suportado em todas as plataformas.
- LinkOption.NOFOLLOW_LINKS
Indica que links simbólicos não devem ser seguidos, apenas copiados.

Exemplo:

```
java
import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;

try {
    Files.copy(Paths.get("in.txt"),
               Paths.get("out.txt"),
               REPLACE_EXISTING);
} catch (IOException e) { /** ... */ }
```

Também existem métodos para copiar entre um fluxo (stream) e um Path:

```
java
static long copy(InputStream in, Path target,
                  CopyOption... options) throws IOException
```

Copia todos os bytes de um fluxo de entrada para um arquivo. Por padrão, a cópia falha se o arquivo de destino já existir ou for um link simbólico. Se a opção StandardCopyOption.REPLACE_EXISTING for especificada e o arquivo de destino já existir, então ele será substituído, se não for um diretório não vazio. Se o arquivo de destino existir e for um link simbólico, então o link simbólico é substituído. Na verdade, no Java 8, a opção REPLACE_EXISTING é a única exigida por este método.

```
java
static long copy(Path source,
                  OutputStream out) throws IOException
```

Copia todos os bytes de um arquivo para um fluxo de saída.

Exemplo:

```
java Copiar Editar

try (InputStream in = new FileInputStream("in.csv");
    OutputStream out = new FileOutputStream("out.csv")) {
    Path path = Paths.get("/temp/in.txt");
    // Copia os dados do fluxo para um arquivo
    Files.copy(in, path);
    // Copia os dados do arquivo para um fluxo
    Files.copy(path, out);
} catch (IOException e) { /** ... */ }
```

Para mover ou renomear um arquivo/diretório, temos o método:

```
java Copiar Editar

static Path move(Path source, Path target,
    CopyOption... options) throws IOException
```

Por padrão, este método seguirá links, lançará uma exceção se o arquivo já existir e **não** realizará uma movimentação atômica.

Podemos personalizar esse comportamento com os seguintes enums CopyOption:

- StandardCopyOption.REPLACE_EXISTING
Executa a movimentação quando o destino já existe. Se o destino for um link simbólico, somente o link é movido.
- StandardCopyOption.ATOMIC_MOVE
Executa a movimentação como uma operação atômica. Se o sistema de arquivos não suportar uma movimentação atômica, uma exceção será lançada.

Este método pode mover um diretório não vazio. No entanto, se o destino existir, tentar mover um diretório não vazio lançará uma DirectoryNotEmptyException. Esta exceção também será lançada ao tentar mover um diretório não vazio entre unidades ou partições.

Exemplo:

```
java Copiar Editar

try {
    // Mover ou renomear dir1 para dir2
    Files.move(Paths.get("c:\\temp\\dir1"),
        Paths.get("c:\\temp\\dir2"));
} catch (IOException e) { /** ... */ }
```

Gerenciando metadados

Ao falar de um sistema de arquivos, metadados nos fornecem informações sobre um arquivo ou diretório, como seu tamanho, permissões, data de criação, etc. Essas informações são chamadas de **atributos**, e alguns deles são dependentes do sistema.

A classe Files possui alguns métodos para obter ou definir alguns atributos de um objeto Path:

```
java Copiar Editar

static long size(Path path) throws IOException
```

Retorna o tamanho de um arquivo (em bytes).

```
java Copiar Editar

static boolean isDirectory(Path path, LinkOption... options)
```

Testa se um arquivo é um diretório.

```
java Copiar Editar

static boolean isRegularFile(Path path, LinkOption... options)
```

Testa se um arquivo é um arquivo comum.

```
java Copiar Editar

static boolean isSymbolicLink(Path path)
```

Testa se um arquivo é um link simbólico.

```
java Copiar Editar

static boolean isHidden(Path path) throws IOException
```

Informa se um arquivo é considerado oculto.

```
java Copiar Editar

static FileTime getLastModifiedTime(Path path,
    LinkOption... options) throws IOException

static Path setLastModifiedTime(Path path,
    FileTime time) throws IOException
```

Retorna ou atualiza o tempo da última modificação de um arquivo.

```
java Copiar Editar

static UserPrincipal getOwner(Path path,
    LinkOption... options) throws IOException

static Path setOwner(Path path,
    UserPrincipal owner) throws IOException
```

Retorna ou atualiza o proprietário do arquivo.

Nos métodos que aceitam opcionalmente `LinkOption.NO_FOLLOW_LINKS`, os links simbólicos **não** são seguidos (por padrão, **são**).

No caso de `getLastModifiedTime()` e `setLastModifiedTime()`, a classe `java.nio.file.attribute.FileTime` representa o valor de um atributo de marca de tempo do arquivo.

Podemos criar uma instância de `FileTime` com os métodos estáticos:

```
java Copiar Editar

static FileTime from(Instant instant)
static FileTime from(long value, TimeUnit unit)
static FileTime fromMillis(long value)
```

E de um `FileTime` podemos obter um `Instant` ou milissegundos como `long`:

java Copiar Editar

```
Instant toInstant()
long toMillis()
```

Por exemplo:

java Copiar Editar

```
try {
    Path path = Paths.get("/temp/dir1/file.txt");
    FileTime ft = Files.getLastModifiedTime(path);
    Files.setLastModifiedTime(path,
        FileTime.fromMillis(ft.toMillis() + 1000)); // adiciona um segundo
} catch (IOException e) { /** ... */ }
```

No caso de `getOwner()` e `setOwner()`, a interface `java.nio.file.attribute.UserPrincipal` é uma representação abstrata de uma identidade que pode ser usada assim:

java Copiar Editar

```
try {
    Path path = Paths.get("/temp/dir1/file.txt");
    // FileSystems.getDefault() também obtém um objeto FileSystem
    UserPrincipal owner = path.getFileSystem()
        .getUserPrincipalLookupService()
        .lookupPrincipalByName("jane");
    Files.setOwner(path, owner);
} catch (IOException e) { /** ... */ }
```

Esses métodos são úteis para obter ou atualizar um único atributo. Mas também podemos obter um grupo de atributos relacionados por funcionalidade ou por uma implementação específica do sistema, como uma visualização (*view*).

As três classes de visualização mais comuns são:

- `java.nio.file.attribute.BasicFileAttributeView`
Fornece uma visualização de atributos básicos suportados por todos os sistemas de arquivos.
- `java.nio.file.attribute.DosFileAttributeView`
Estende `BasicFileAttributeView` para suportar adicionalmente um conjunto de flags de atributos DOS usados para indicar se o arquivo é somente leitura, oculto, um arquivo de sistema ou arquivado.
- `java.nio.file.attribute.PosixFileAttributeView`
Estende `BasicFileAttributeView` com atributos suportados em sistemas POSIX, como Linux e Mac. Exemplos desses atributos são proprietário do arquivo, grupo proprietário e permissões de acesso relacionadas.

Você pode obter uma visualização de atributos de arquivo de um tipo dado para ler ou atualizar um conjunto de atributos com o método:

java Copiar Editar

```
static <V extends FileAttributeView> V getFileAttributeView(
    Path path, Class<V> type, LinkOption... options)
```

Por exemplo, `BasicFileAttributeView` tem apenas um método de atualização:

```

java Copiar Editar

try {
    Path path = Paths.get("/temp/dir/file.txt");
    BasicFileAttributeView view =
        Files.getFileAttributeView(path,
            BasicFileAttributeView.class);
    // Obtém uma classe com atributos somente leitura
    BasicFileAttributes readOnlyAttrs =
        view.readAttributes();
    FileTime lastModifiedTime =
        FileTime.from(Instant.now());
    FileTime lastAccessTime =
        FileTime.from(Instant.now());
    FileTime createTime =
        FileTime.from(Instant.now());
    // Se qualquer argumento for null,
    // o valor correspondente não será alterado
    view.setTimes(lastModifiedTime,
        lastAccessTime,
        createTime);
} catch (IOException e) { /** ... */ }

```

Na maioria das vezes, você trabalhará com as versões somente leitura das visualizações de arquivos. Neste caso, você pode usar o seguinte método para obtê-las diretamente:

```

java Copiar Editar

static <A extends BasicFileAttributes> A
    readAttributes(Path path, Class<A> type,
        LinkOption... options)
    throws IOException

```

O segundo parâmetro é o tipo de retorno do método, a classe que contém os atributos a serem usados (observe que todas as classes de atributos estendem `BasicFileAttributes` porque ela contém atributos comuns a todos os sistemas de arquivos). O terceiro argumento é usado quando você deseja seguir links simbólicos.

Aqui está um exemplo de como acessar os atributos de um arquivo usando a classe `java.nio.file.attribute.BasicFileAttributes`:

```

java Copiar Editar

try {
    Path path = Paths.get("/temp/dir1/file.txt");
    BasicFileAttributes attr = Files.readAttributes(
        path, BasicFileAttributes.class);
    // Tamanho em bytes
    System.out.println("size(): " + attr.size());
    // Identificador único do arquivo (ou null se não estiver disponível)
    System.out.println("fileKey(): " + attr.fileKey());

    System.out.println("isDirectory(): " + attr.isDirectory());
    System.out.println("isRegularFile(): " + attr.isRegularFile());
    System.out.println("isSymbolicLink(): " + attr.isSymbolicLink());
    // É algo diferente de arquivo, diretório ou link simbólico?
    System.out.println("isOther(): " + attr.isOther());

    // Os métodos a seguir retornam uma instância de FileTime
    System.out.println("creationTime(): " + attr.creationTime());
    System.out.println("lastModifiedTime(): " + attr.lastModifiedTime());
    System.out.println("lastAccessTime(): " + attr.lastAccessTime());
} catch (IOException e) { /** ... */ }

```

Pontos-chave

- As classes principais de `java.nio.file` são `Path`, `Paths` e `Files`. Elas têm como objetivo substituir a classe `java.io.File`.
- A interface `Path` define um objeto que representa o caminho para um arquivo ou diretório.
- `java.nio.file.Paths` fornece métodos para criar um objeto `Path`.
- A representação de caminho absoluto de um objeto `Path` possui um componente raiz (ou `c:\` ou `/`) e uma sequência de nomes separados por uma barra (normal ou invertida).
- A interface `Path` possui métodos para obter os elementos do caminho, normalizar caminhos e obter atributos do caminho (`isAbsolute()`, `getFileSystem()`, etc.), entre outros.
- Também implementa `Comparable` e `equals()` para teste de igualdade.
- A classe `java.nio.file.Files` possui métodos estáticos para operações comuns em arquivos e diretórios. Em contraste com `java.io.File`, todos os métodos de `Files` funcionam com objetos `Path`.
- Exemplos dessas operações incluem verificar existência de arquivo, copiar, mover, excluir e ler.
- Você também pode obter atributos de um arquivo individualmente (com métodos como `isHidden()`) ou em grupo por meio de visualizações.
- As três classes de visualização mais comuns são `BasicFileAttributeView`, `DosFileAttributeView` e `PosixFileAttributeView`.
- Você pode obter uma visualização de atributos de arquivo de um tipo específico com o método `getFileAttributeView()`.
- Você pode obter uma classe que é uma versão somente leitura da visualização com o método `readAttributes()`.

Autoavaliação

1. Dado:

```
java
Path path1 = Paths.get("/projects/work/../../fun");
Path path2 = Paths.get("games");
System.out.println(path1.resolve(path2));
```

Qual das seguintes é o resultado da execução das linhas acima?

- A. `/project/work/fun/games`
 - B. `/project/fun/games`
 - C. `/project/work/../../fun/games`
 - D. `games`
-

2. Dado:

```
java
Path path = Paths.get("c:\\Users\\mark");
```

Qual das seguintes retornará Users?

- A. path.getRoot()
 - B. path.getName(0)
 - C. path.getName(1)
 - D. path.subpath(0, 0)
-

3. Qual das seguintes não é uma opção válida de CopyOption para Files.copy()?

- A. NOFOLLOW_LINKS
 - B. REPLACE_EXISTING
 - C. ATOMIC_MOVE
 - D. COPY_ATTRIBUTES
-

4. Dado:

```
java
Path path =
    Paths.get("c:\\.\\temp\\data\\...\\dir\\...\\file.txt");
try {
    path = path.toRealPath();
} catch (IOException e) { }
System.out.println(path.subpath(1,2));
```

Qual é o resultado?

- A. temp
 - B. data
 - C. dir
 - D. file.txt
-

5. Qual das seguintes é uma maneira válida de definir o tempo de criação de um arquivo?

A.



```
java
FileTime time = FileTime.from(Instant.now());
Files.getFileAttributeView(path,
    BasicFileAttributeView.class)
    .setTimes(null, time, null);
```

B.

```
java
Files.setCreateTime(path,
    FileTime.from(Instant.now()));
```

C.

java

 Copiar  Editar

```
Files.getFileAttributeView(path,  
    BasicFileAttributeView.class)  
    .setTimes(null, null, Instant.now());
```

D.

java

 Copiar  Editar

```
FileTime time = FileTime.from(Instant.now());  
Files.getFileAttributeView(path,  
    BasicFileAttributeView.class)  
    .setTimes(null, null, time);
```